Throughput optimization for micro-factories subject to task and machine failures

Anne Benoit, Alexandru Dobrila, Laurent Philippe, Jean-Marc Nicod

Throughput optimization for micro-factories subject to task and machine failures

Anne Benoit
ENS Lyon, Université de Lyon, France
LIP laboratory (ENS, CNRS, INRIA, UCBL)
Anne.Benoit@ens-lyon.fr

Alexandru Dobrila, Jean-Marc Nicod and Laurent Philippe
Laboratoire d'Informatique de Franche-Comté,
Université de Franche-Comté, France
adobrila,jmnicod,lphilippe@lifc.univ-fcomte.fr

***Abstract*—In this paper, we study the problem of optimizing the throughput for micro-factories subject to failures. The challenge consists in mapping several tasks of different types onto a set of machines. The originality of our approach is the failure model for such applications in which not only the machines are subject to failures but the reliability of a task may depend on its type. The failure rate is unrelated: a probability of failure is associated to each couple (task type, machine). We consider different kind of mappings: in one-to-one mappings, each machine can process only a single task, while several tasks of the same type can be processed by the same machine in specialized mappings. Finally, general mappings have no constraints. The optimal one-to-one mapping can be found in polynomial time for particular problem instances, but the problem is NP-hard in most of the cases. For the most realistic case of specialized mappings, which turns out to be NP-hard, we design several polynomial time heuristics and a linear program allows us to find the optimal solution (in exponential time) for small problem instances. Experimental results show that the best heuristics obtain a good throughput, much better than the throughput achieved with a random mapping. Moreover, we obtain a throughput close to the optimal solution in the particular cases where the optimal throughput can be computed.**

I. Introduction

In this paper, we study the problem of optimizing the throughput for micro-factories subject to failures. Micro-factories are production systems composed of cells, each one performing a particular task on complex micro-components that pass through them. The probability for a fault to arise in these cells is high, so taking faults into account is mandatory when scheduling a production. In this context, faults are however not only attached to the processing unit, as it is commonly assumed for computer based distributed systems, but also to the tasks. In a production system a task may indeed be complex to perform, for instance due to some hard manipulation, with an impact on its success ratio. If the same robot is able to perform different tasks, it may generate less faults on simple tasks than on difficult ones.

To produce a micro-product, several tasks, each characterized by a task type, must be performed by the cells in an order fixed by a precedence graph. In the micro-factory, the robots that compose the cells must however be configured before being able to process a type of task. So the issue we face is to map several tasks of different types onto a set of cells, or machines, with the objective of optimizing the number of products that output the system, in spite of the faults.

In a first study [1], we have tackled the particular case in which faults only depend on the task type. In this paper we are interested in studying the impact of a fault model linked to both tasks and machines. Our specific use case is a micro-factory, more a production system than a distributed computing system, but the results presented in this paper are more generally applicable to distributed production systems or to distributed systems where the fault probability is attached not only to resources, but also to tasks.

The paper is organized as follows. The micro-factory context and related works are presented in Section II. Section III gives a more formal presentation of the micro-factories and of the failure model. Section IV presents the optimization problems tackled in the paper. The complexity study and results are given in Section V. In the rest of the paper, we focus on a particular variant of the problem, which is NP-hard: our aim is to find a specialized mapping which maximizes the throughput of a linear chain application. In Section VI, we provide several methods to solve this problem: (i) an integer linear programming formulation of the problem which allows us to find the optimal solution for small problem instances, and (ii) polynomial time heuristics for general instances. An extensive set of simulations is detailed in Section VII, and demonstrates the efficacy of our heuristics. Finally, we conclude in Section VIII.

II. Context and state of the art

Micro-factories are production units designed to produce pieces composed of micro-metric elements [2]. Today's micro-factories are composed of micro-robots able to carry out basic operations through elementary actuators as piezo-electric beams (e.g., for gripping), stick-slip systems, etc. As these robots are usually teleoperated by a human operator, only simple tasks can be done. To perform more complex operations and to improve their efficiency, micro-factories need to be automated and robots need to be grouped in cells. Then cells will be put together and they will cooperate to produce complex assembled pieces, as it is done for macroscopic productions. Due to the piece, actuator and cell sizes, it is however impossible for human operators to directly interfere with the physical system. So it needs a highly automated command. The complexity of this command makes it mandatory to develop a distributed system to support this control. So, the

cell group results in a distributed system that is very similar to a distributed computing platform. However, at this scale the physical constraints are not totally controlled so there is a need to take faults into account in the automated command.

The main issue for fault tolerant systems [3] is to overcome the failure of a node, a machine or a processor. To deal with those faulty machines, the most common method used in distributed systems is to replicate [4] the data. Those models assume that failures are attached to a machine. So the probability to get one product as a result is highly increased when the task is replicated on several machines. Once all the replicated jobs are done, a vote algorithm [5] is often used to decide which result is the right one. However, in our case the products are physical objects and therefore can not be replicated.

In real-time systems, another model called Window-Constrained [6] model can be used. In this model one considers that, for y messages, only x ($x \leq y$) of them will reach their destination. The y value is called the Window. The looses are not considered as a failure but as a guarantee: for a given network a Window-Constrained Scheduling [7], [8] can guarantee that no more than x messages will be lost for every y sent messages. The Window-Constrained based failure model is adapted to a distributed system, the micro-factory. But in this paper, the objective function makes us use the failure model as the ratio x/y. In any case, the issue is to guarantee the output of a given number of products. Once an allocation of tasks to machines has been given, we can compute the number of products needed as input of the system and guarantee the output for the desired number of products.

III. Framework

We outline in this section the characteristics of the applicative framework and target platform. Finally, we describe and motivate the failure model that we use in this work.

A. Applicative framework

We consider a set $\mathcal{N}$ of n tasks: $\mathcal{N} = \{T_1, T_2, \ldots, T_n\}$. Each task T_i ($1 \leq i \leq n$) is applied successively on a set of products. We wish to produce x_{out} products as an output, and the total number of products being processed by a task may depend on the allocation: we process more than x_{out} products since some losses may occur because of failures, as explained later in Section III-C. Note that all products are identical. When the context is not ambiguous, we may also design task T_i by i for clarity, as for instance in the figures.

A type is associated to each task as the same operation may be applied several times to the same product. Thus, we have a set $\mathcal{T}$ of p task types with $n \geq p$ and a function $t : [1..n] \to \mathcal{T}$ which returns the type of a task: $t(i)$ is the type of task T_i, for $1 \leq i \leq n$.

The application is a directed acyclic graph (DAG) in which the vertices are tasks, and edges represent dependencies between tasks. An example of application with $n = 5$ tasks is represented on Figure 1. In the top branch of the DAG, we need to finish the processing of task T_1 on one product before proceeding to task T_2. The join to task T_4 corresponds to the merge of two products, which produces a new unit of product composed of the two. Typically one instance of product from each predecessor in the graph is required to process with the joining task. Note that forks cannot be considered in this context as the output of one task is a physical component that cannot be split in two. Unlike data that can be easily replicated at every step of a DAG, an instance of a physical component is the result of all the preceding tasks and cannot be duplicated as it is material.

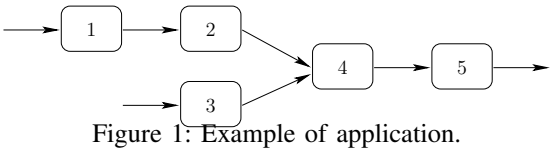

Figure 1: Example of application.

B. Target platform

The platform consists in a set $\mathcal{M}$ of m machines: $\mathcal{M} = \{M_1, M_2, \ldots, M_m\}$. All machines can be interconnected: the platform graph is a complete graph. A machine handles some of the tasks at a given speed: machine M_u can perform the task T_i onto one product in a time $w_{i,u}$. We also consider that tasks of the same type have the same execution time on a given machine, since they correspond to the same action to be performed on the products. Thus, we have:

$$\forall i, i' \in [1, n], \forall u \in [1, m], t(i) = t(i') \Rightarrow w_{i,u} = w_{i',u}.$$

We neglect the communication time required to transfer a product from one machine to another. If a communication may not be negligible, we can always model it as a particular task with a dedicated machine (the machine responsible of the transfer of the product).

We are interested in producing the desired number of products rather than producing a particular instance of a product. So we consider that products are not identified: two products, on which the same sequence of tasks has been done, are exactly similar and we can use one or the other indifferently for further operations.

C. Failure model

An additional characteristic of our framework is that tasks are subject to failure. It may happen that a product is lost or damaged while a task is being executed on this product. For instance electrostatic strength may be accumulated on the actuator, and thus the piece will be pushed away rather than caught. Indeed, we work at a scale such that these electrostatic strengths are stronger than gravity.

Due to our application setting, we deal only with transient failures, as defined in [9]. The tasks are failing for some of the products, but we do not consider a permanent failure of the machine responsible of the task, as this would lead to a failure for all the remaining products to be processed and the unability to finish them.

One classical technique used to deal with failures is replication [4]. However, while replication is very useful for hardware failures of machines, we cannot use it in our framework since

the products are not a data such as a numerical image that we need to process, but it is a physical object. The cost of these products is very low while the equipments are expensive. Thus, the only solution consists in processing more products than needed, so that at the end, the required number of finished products are brought out.

The failure rate of task T_i performed onto machine M_u is the percentage of failure for this task and is denoted $f_{i,u} = \frac{l_{i,u}}{b_{i,u}}$, where $l_{i,u}$ is the number of lost products each time $b_{i,u}$ products have been processed ($l_{i,u} \leq b_{i,u}$).

IV. Optimization problems

Now that the framework has been clarified, we formalize in this section the various optimization problems that we wish to solve. Our goal is to assign tasks to machines so as to optimize some key performance criteria. The solution to one problem is thus an allocation function $a : [1..n] \to [1..m]$ which returns for each task the machine on which it is executed. Thus, if $a(i) = u$, task T_i is executed on machine M_u, and the processing of one product for this task takes a time $w_{i,u}$.

We first discuss the objective criteria that we want to optimize. Then we introduce the different rules of the game that can be used in the definition of the allocation function a. The complexity of these various problems is discussed in Section V.

A. Objective function

In our framework, several objective functions could be optimized. For instance, one may want to produce a mapping of the tasks on the machines as reliable as possible, i.e., minimize the total number of products to input in the system. Rather, we consider that products are cheap, and we focus on a performance criteria, the throughput. The goal is to maximize the number of products processed per time unit, making abstraction of the initialization and clean-up phases. This objective is important when a large number of products must be produced.

Rather than maximizing the throughput of the application, we rather deal with the *period*, which is the inverse of the throughput. First we introduce the fractional number x_i, which is the average number of products required to output one product out of the system for task T_i. We can compute x_i recursively for any application. Let T_j be the (unique) successor of T_i, if it exists (remember that we do not allow forks in the application graph). For tasks with no successor, we set $x_j = 1$, which means that T_i needs to output one product. Then, if task T_i is assigned to machine M_u, we have

$$x_i = \frac{1}{1 - f_{i,u}} \times x_j = \frac{b_{i,u}}{b_{i,u} - l_{i,u}} \times x_j \ ,$$

where the fraction represents the number of products needed per successful product. Starting from the nodes with no successor, we can then compute x_i for each task T_i.

We are now ready to define the *period* of a machine: it is the time needed by a machine to execute all the tasks allocated onto this machine in order to produce one final product out of the system. Formally, we have

$$period(M_u) = \sum_{1 \leq i \leq n | a(i) = u} x_i w_{i,u} \ . \tag{1}$$

The period of machine M_u is the sum, for each task allocated to that machine, of the average number of products (x_i) needed to output one product, multiplied by the speed ($w_{i,u}$) of that task onto that machine. The slowest machine will slow down the whole application, thus we aim at minimizing the largest machine period. The machines realizing this maximum are called *critical machines*. If M_c is a critical machine, then $period = period(M_c) = \max_{M_u \in \mathcal{M}} period(M_u)$.

Note that minimizing the period is similar to maximizing the throughput.

B. Rules of the game

In this section, we classify several variants of the optimization problem that has been introduced. A task must always be processed by one unique machine (allocation function), but different rules can be enforced about what a machine can process.

1) One-to-one mappings: In this first class of problems, a machine can compute only one single task. This rule of the game is enforced with the following constraint, meaning that a machine cannot compute two different tasks:

$$\forall 1 \leq i, i' \leq n \quad i \neq i' \Rightarrow a(i) \neq a(i') \ .$$

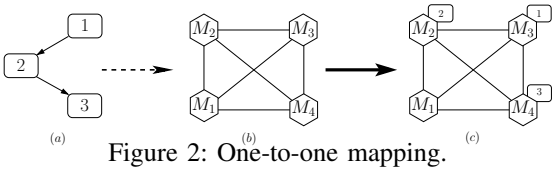

Figure 2: One-to-one mapping.

On Figure 2, we have an application graph (a) that must be mapped on a platform graph (b). The result is shown in (c), where we can see that one machine can handle only one task. Thus this mapping is quite restrictive because we must have at least as many machines as tasks.

2) Specialized mappings: We have dedicated machines that can realize only one type of tasks. But task types are not dedicated to machines, so two machines may compute different tasks of the same type.

For instance, let us consider five tasks T_1, T_2, T_3, T_4, T_5 with the following types: $t(1) = t(3) = t(5) = 1$ and $t(2) = t(4) = 2$. If machine M_3 computes task T_1, it could also execute T_3 and T_5 but not T_2 and T_4. As types are not dedicated to machines, T_5 could also be assigned to another machine, for instance M_1. This situation is described on Figure 3.

The following constraint expresses the fact that a machine cannot compute two tasks of different types:

$$\forall 1 \leq i, i' \leq n \quad t(i) \neq t(i') \Rightarrow a(i) \neq a(i') \ .$$

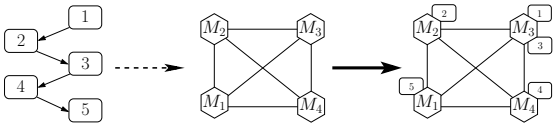

Figure 3: Specialized mapping (task types: $t(1) = t(3) = t(5) = 1$ and $t(2) = t(4) = 2$).

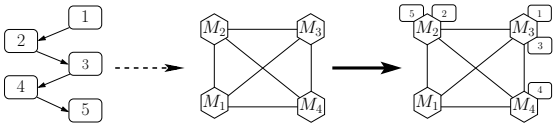

Figure 4: General mapping (task types: $t(1) = t(3) = 1$, $t(2) = t(4) = 2$ and $t(5) = 3$).

3) General mappings: A machine can compute any task regardless of its type, thus there are no constraints. An example of this case is shown on Figure 4.

V. Complexity results

Complexity results are classified depending on the mapping rules. We start with one-to-one mappings, then we focus on specialized and general ones.

A. Complexity of one-to-one mappings

For one-to-one mappings, we can refine the problem complexity depending on the application class. We are particularly interested in linear chain applications, as in the example of Figure 1. Indeed, the problem remains polynomial for such applications (Theorem 1), while it turns out to be NP-hard for general applications (Theorem 2).

In this section, we introduce a new notation: $F_i = \frac{1}{1-f_{i,a(i)}}$.

Theorem 1. *Given a linear chain application and a set of homogeneous machines ($w_{i,u} = w$ for all i, u), finding the one-to-one mapping which maximizes the throughput can be done in polynomial time.*

Proof: For a linear chain application with dependencies from task T_i to task T_{i+1}, for $1 \leq i \leq n-1$, the average number of products x_i needed to output one product out of the system and performed by the task T_i can be computed thanks to the F_j, with $j \geq i$, see Section IV-A:

$$x_i = F_i \times x_{i+1} = \prod_{i \leq j \leq n} F_j \ . \tag{2}$$

Of course, the values of F_j depend on the allocation function. Thus, the period $period(M_{a(i)}) = period(i)$ of the machine $M_{a(i)}$ on which the task T_i is assigned to is $x_i \times w_{i,a(i)} = x_i \times w$. Since all F_j values are greater than 1, we have $x_1 = \max_{1 \leq i \leq n} x_i$, and the period is constrained by the machine on which task T_1 is executed. The goal is thus to minimize the product $\prod_{1 \leq j \leq n} F_j$, in order to maximize the period.

Since the mapping is required to be one-to-one, we create a bipartite graph with one node per task T_j ($1 \leq j \leq n$) on one side, one node per machine on the other side. The cost of an edge from task T_j to machine M_u is then set to $-\log(1-f_{j,u})$, so as to transform the previous product into a sum.

Then, we can find in polynomial time a minimum weight matching in this bipartite graph, for instance using the Hungarian algorithm [10], [11]. This matching corresponds to an assignment of tasks to machines which minimizes $\prod_{1 \leq j \leq n} F_j$, and thus it is equivalent to a one-to-one mapping which has a minimum period.

■

Note that this reasoning does not hold anymore with heterogeneous machines (w_i, w_u or $w_{i,u}$), since the bottleneck task is not necessarily T_1 in such cases. The complexity remains open for such cases.

However, if we consider general applications rather than restricting to linear chains, the problem becomes NP-hard.

Theorem 2. *Finding the optimal one-to-one mapping is NP-hard, even with constant processing costs w and failure rates which depend on machines ($f_{i,u} = f_u$ for $1 \leq i \leq n$).*

Proof: We consider the following decision problem: given a period K, is there a one-to-one mapping whose period does not exceed K? The problem is obviously in NP: given a period and a mapping, it is easy to check in polynomial time whether it is valid or not.

The NP-completeness is obtained by reduction from 3-PARTITION [12], which is NP-complete in the strong sense. Let $\mathcal{I}_1$ be an instance of 3-PARTITION: given a set $\{z_1, ..., z_{3n}\}$ of $3n$ integers, and an integer Z such that $\sum_{1 \leq j \leq 3n} z_j = nZ$, does there exist n independent subsets $B_1, \cdots, B_n$ of $\{z_1, \cdots, z_{3n}\}$ such that for all $1 \leq i \leq n$, $\sum_{z_j \in B_i} z_j = Z$?

We build the following instance $\mathcal{I}_2$ with $3n+1$ tasks and processors:

- the application consists in n linear chains of 4 tasks sharing the same final task $T^{(4)}$: for $1 \leq i \leq n$, $T_i^{(1)} \rightarrow T_i^{(2)} \rightarrow T_i^{(3)} \rightarrow T^{(4)}$;
- $w = 1$ (constant processing cost);
- $f_{3n+1} = 0$ (machine M_{3n+1} never fails);
- for $1 \leq u \leq 3n$, $f_u = \frac{2^{z_u}-1}{2^{z_u}}$;
- $K = 2^Z$.

Note that the size of $\mathcal{I}_2$ is polynomial in the size of $\mathcal{I}_1$. Indeed, since 3-PARTITION is NP-complete in the strong sense, we could encode $\mathcal{I}_1$ in unary, and thus the size of the instance would be in $O(nZ)$. Moreover, the values of f_u can be encoded in binary and thus their size is polynomial in the size of $\mathcal{I}_1$.

Now we show that $\mathcal{I}_1$ has a solution if and only if $\mathcal{I}_2$ has a solution. Suppose that $\mathcal{I}_1$ has a solution. We construct the allocation function a such that $a(T^{(4)}) = 3n+1$, i.e., the last task is processed by the reliable processor, and, for $1 \leq i \leq n$, tasks $T_i^{(1)}, T_i^{(2)}, T_i^{(3)}$ are allocated to the three processors such that $z_u \in B_i$. Since all w are equal to 1, the period of the mapping is constrained by one of the $T_i^{(1)}$ tasks, and their period is $P_i = \prod_{z_u \in B_i} \frac{1}{1-f_u}$. Taking the logarithm, $\log_2(P_i) = \sum_{z_u \in B_i} \log_2(\frac{1}{1-f_u}) = \sum_{z_u \in B_i} \log_2(2^{z_u}) = Z =$

$\log_2(K)$, that means $P_i = K$ for $1 \leq i \leq n$, and $\mathcal{I}_2$ has a solution.

Suppose now that $\mathcal{I}_2$ has a solution. The critical resource is still one of the $T_i^{(1)}$, since $w = 1$. For each of these machines, we must have $\log_2 P_i \leq Z$, and thus $\sum_{u \in alloc_i} z_u = Z$, where $alloc_i$ represents the set of indices of the four processors allocated to the i^{th} chain. To minimize this quantity, we can build a solution in which the reliable processor is processing task $T^{(4)}$, and then the problem amounts to 3-PARTITION the z_u. Therefore, $\mathcal{I}_1$ has a solution. This concludes the proof. ■

This NP-hardness result illustrates the additional difficulty of having a failure probability which depends both on tasks and machines. Indeed, the problem can be solved in polynomial time with fully heterogeneous machines ($w_{i,u}$) when the failure rates are identical for all machines ($f_{i,u} = f_i$ for each machine), because we are then able to compute x_i for each task, independently of the mapping (see [1]).

B. Complexity of specialized and general mappings

In [1], we proved that the problem of finding the optimal specialized or general mapping is NP-hard, even for a linear chain application with constant processing costs w, and when failure probabilities are independent of the machines ($f_{i,u} = f_i$). Therefore, the problem remains NP-hard when considering more general values of failure probabilities. This illustrates the additional complexity of considering more general mapping rules rather than restricting to one-to-one mappings.

VI. Solving the specialized mapping problem

In the practical setting of micro-factories, general mappings are not really useful because of the unaffordable reconfiguration costs. Indeed, if a machine is processing tasks of different types, one needs to reconfigure the machine between operations.

However, when the number m of machines is greater than the number p of task types, it is always possible to find a specialized mapping, since each machine is able to process all the tasks of a same type. The key point is then to find m (or less) groups of tasks of the same type to be assigned to the m machines of the platform. Even if we restrict to specialized mappings and linear chain applications, this problem is already NP-hard, as explained in Section V-B.

Thus, we present in the following a linear program and six heuristics that return a mapping, by grouping tasks of same type onto machines.

A. Linear programming for the specialized mapping

In this section, we present a linear program to solve the specialized mapping problem presented in Section IV-B2. This linear program is a Mixed Integer Program (MIP) because it uses both integer and rational variables. Solving a MIP is NP-complete, however efficient solvers such as Cplex [13] makes it possible to solve small problem instances in a reasonable time. The following MIP implementation allows us to validate the relevance of the scalable heuristics that we present in the next section.

In the following, the two indices i and u denote respectively a task T_i ($1 \leq i \leq n$) and a machine M_u ($1 \leq u \leq m$).

The parameters of the linear program are the following:

- $w_{i,u}$ is the time needed by the task T_i to perform one product onto the machine M_u;
- $f_{i,u}$ is the failure rate of task T_i on machine M_u.

The variables needed to define the MIP are the following:

- x_i is the average number of products that the task T_i has to perform to output one product out of the system;
- For any pair (T_i, M_u) we denote $a_{i,u} \in \{0, 1\}$ as the mapping of T_i onto the machine M_u: $a_{i,u} = 1$ if the task T_i is performed by the machine M_u and 0 otherwise;
- For any pair (M_u, j) we denote $t_{u,j} \in \{0, 1\}$ such as $t_{u,j} = 1$ if the machine M_u is specialized to perform tasks of type j and 0 otherwise;
- $K \in \mathbb{Q}$ is a rational number which represents the upper bound on the period for all machines.

The objective function is to minimize the period K, but several constraints must be enforced to have a valid mapping function (a), and a correct number of product (x).

- We ensure that each task T_i is performed by one and only one machine M_u:

$$\forall i \quad \sum_u a_{i,u} = 1 \tag{3}$$

- We ensure that each machine M_u is dedicated to at most one type j:

$$\forall u \quad \sum_j t_{u,j} \leq 1 \tag{4}$$

- We ensure that each task T_i of type $j = t(i)$ can be performed only by one machine M_u which is specialized upon the type $t(i)$. This constraint is not in contradiction with the fact that several tasks of same type j can be performed by the machine M_u:

$$\forall u \, \forall i \quad a_{i,u} \leq t_{u,t(i)} \tag{5}$$

- We ensure that the average number of products that the task T_i has to perform depends on the mapping of T_i but also on the number of products that the task T_{i+1} has to perform to output one product out of the system.

$$\forall u \, \forall i \quad x_i \geq \frac{1}{1 - f_{i,u}} a_{i,u} \times x_{i+1}$$

This formula can be transformed into the following linear equation:

$$\forall i \, \forall u \quad x_i \geq \frac{1}{1 - f_{i,u}} \, x_{i+1} - (1 - a_{i,u}) MAX_{x_i} \tag{6}$$

where MAX_{x_i} is an upper bound of x_i such that $x_i \leq MAX_{x_i} = \prod_{i \leq j \leq n} \frac{1}{1 - \max_{1 \leq u \leq m}(f_{j,u})}$.

- The period of each machine M_u depends on the mapping and its value is bounded by K:

$$\forall u \quad \sum_i a_{i,u} \times x_i w_{i,u} \leq K$$

This non-linear formula can be transformed into the following linear inequations. In order to make the linearization possible, we define a new positive rational variable $y_{i,u} = a_{i,u} \times x_i$ for every task T_i and for every machine M_u. So the previous equation can be rewritten into the equation (7) under the constraints (8):

$$\forall u \quad \sum_i y_{i,u} w_{i,u} \leq K \tag{7}$$

$$\begin{cases} \forall i \, \forall u & y_{i,u} \leq a_{i,u} MAX_{x_i} \\ \forall i \, \forall u & y_{i,u} \leq x_i \\ \forall i \, \forall u & y_{i,u} \geq x_i - (1 - a_{i,u}) MAX_{x_i} \end{cases} \tag{8}$$

The objective is to minimize the period under the previous constraints, thus we get the following MIP:

$$\begin{cases} \text{Minimize } K \\ \text{under the constraints } (3), (4), (6), (7), (8) \end{cases} \tag{9}$$

B. Heuristics

Since faults occur depending on the task and the machine, we are not able to compute the number of products the task T_i has to perform before knowing which task is assigned to which machine. The six heuristics presented here are executed by starting with the last task of the application graph and going backward to the first one.

***H1**: Random heuristic*. A task T_i is assigned to a machine M_u if M_u is free or if M_u is already specialized to tasks of the type $t(i)$. If none of these conditions are fulfilled, we try the next machine M_{u+1} and so on until an available machine is found.

***H2**: Binary search heuristic 1*. This heuristic aims at optimizing the potential of the machines, i.e., the goal is to assign to each machine a set of tasks for which it is efficient. Thus, we start by sorting, for each machine M_u, the set of $w_{i,u}$, for $1 \leq i \leq n$, in ascending order. Then, $rank_{i,u}$ represents the rank of T_i in the ordered set for M_u.

The heuristic performs a binary search on the period between 0 (best case) and the time required to perform sequentially all the tasks on a machine (worst case). For each value of the search, all tasks are assigned greedily (from T_1 to T_n) to machines.

We try to assign the task T_i to a machine such that $rank_{i,u}$ is minimum. If the rank equals one, this means that the potential of M_u for this task is optimal. In case of equality (several machines of identical rank for T_i), machines are sorted by non-decreasing values of $w_{i,u}$. Of course, the assignment can be done only if the machine was not already specialized to a type which is different from $t(i)$, and if the fixed period is not exceeded. Otherwise we try to assign T_i to the next machine, according to their priority order for this task. If no machine is able to process T_i, then no assignment is found and we try a larger period. If all tasks can be correctly assigned, we try a smaller period.

***H3**: Binary search heuristic 2*. This heuristic is the same as H2 except that, for the assignment, the machines are sorted by their heterogeneity level in descending order. The idea is to preserve homogeneous machines for the last tasks. The heterogeneity level of M_u is computed as the standard deviation of its $w_{i,u}$ values. Each task is assigned to the most heterogeneous machine capable of handling it. Note that for this heuristic, slow machines may be used instead of powerful ones, because of their heterogeneity level.

***H4**: Best performance heuristic*. This heuristic assign a task T_i to the machine M_u with the best performance value for that task. The performance value of M_u for T_i is computed by $w_{i,u} \times f_{i,u} \times x_i$.

***H4w**: Faster machine heuristic*. This heuristic is the same that H4 except that the faster machine is selected ($w_{i,u} \times x_i$) without taking into account the failure rate in the assignment process.

***H4f**: Reliable machine heuristic*. This heuristic is the same that H4 except that the most reliable machine is selected ($f_{i,u} \times x_i$) without taking into account the speed in the assignment process.

VII. Experiments

In this section, we compare the six heuristics that give scheduling solutions to the specialized mapping problem with $w_{i,u}$ and $f_{i,u}$ for linear chain applications. The results are computed by a simulator, developped in C++. The performance of each heuristic is measured by its period in ms.

Recall that m is the number of machines, p the number of types, and n the number of tasks. Each point in the figures is an average value of 30 simulations where the $w_{i,u}$ are randomly chosen between 100 and 1000 *ms*, for $1 \leq i \leq n$ and $1 \leq u \leq m$. Similarly, failure rates $f_{i,u}$ are randomly chosen between 0.5% and 2% (i.e., 1/200 and 1/50), unless stated otherwise.

A. Specialized mappings with m and p fixed

In this first set of experiments, the number of machines m and the number of task types p are fixed, and we plot the period for each heuristic as a function of the number of tasks n.

Figure 5 shows that H1 and H4f are not very competitive. Indeed, minimizing the failure rate does not prevent from choosing a slow machine and so getting a long period. For the next experiment, only the other heuristics are plotted. In Figure 6, H4 is slightly under the others. That is explained by the $f_{i,u}$ factor used by H4. Two major factors are in competition here, the speed and the reliability. A large platform is set (100 machines) to see the difference between those two factors. In Figure 7, H4w shows up to be better than the others. As a conclusion of this first set of experiments, we can say that the machine speed seems to be a more important criteria than its reliability when taking assignment choices.

To study more precisely the effect of failure rates, platforms with a high failure rate (up to 10%) are used. Figure 8 shows that periods are increasing dramatically with the number of tasks. In that special case only H2 is performing well.

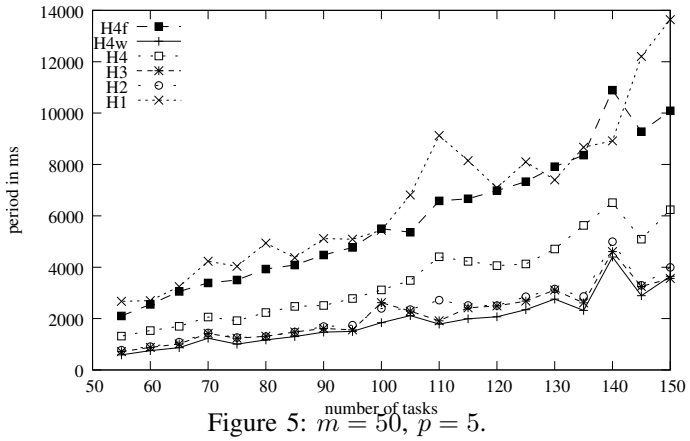

Figure 5: $m = 50$, $p = 5$.

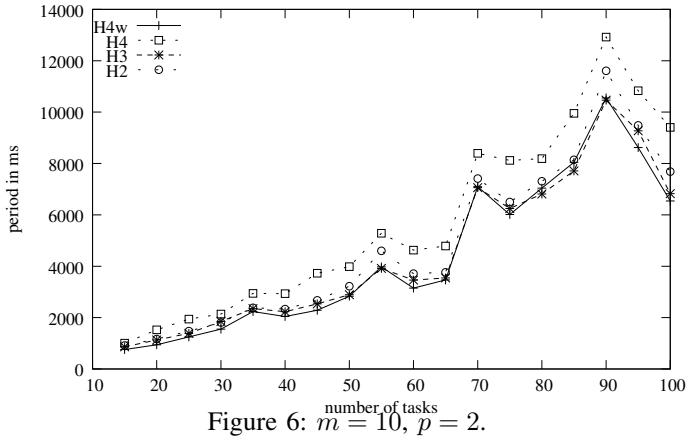

Figure 6: $m = 10$, $p = 2$.

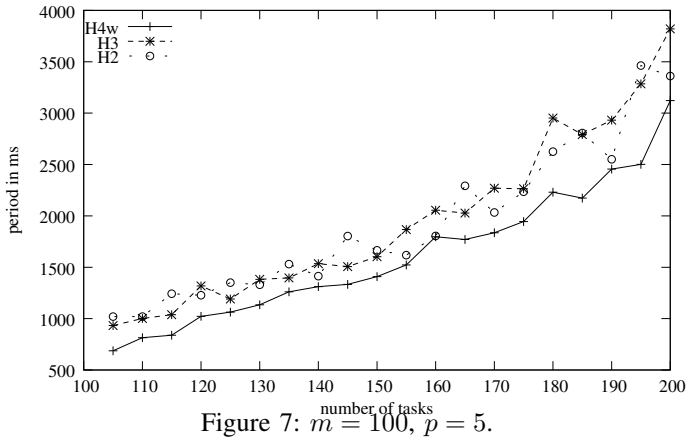

Figure 7: $m = 100$, $p = 5$.

B. One-to-one mappings with m and n fixed

As shown in section V-A, considering the one-to-one mapping, an optimal solution can be found in polynomial time only if the failure is attached only to tasks ($f_{i,u} = f_i$ for $1 \leq u \leq m$). Thus, a platform with 100 machines, 100 tasks and failures defined by f_i is set. We plot the period as a function of the number of types p and run 100 simulations for each dot of the figure. Figure 9 shows H2, H3, H4w and the optimal one-to-one solution (OtO). For a better visibility the other heuristics are ignored here. H4w has the best performance and is very close to the optimal when the number of types is low. We can also see that when the number of types is high, all heuristics tend to have the same performance. This is explained by the fact that with p close to m, the way of creating the groups of tasks is less crucial. Results are very encouraging and show that H2, H3 and H4w are respectively at a factor of 1.84, 1.75 and 1.28 from the optimal solution.

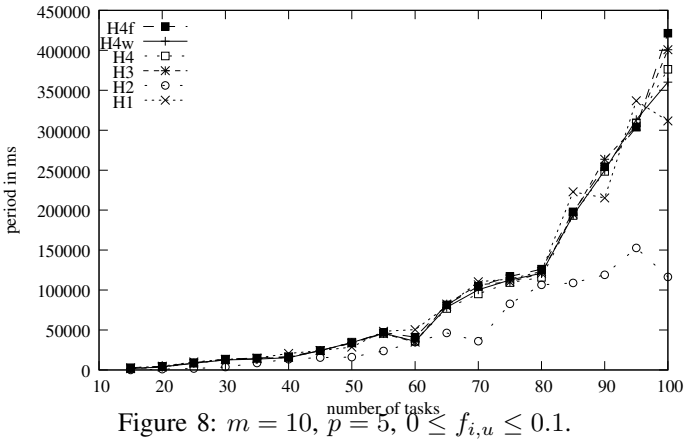

Figure 8: $m = 10$, $p = 5$, $0 \leq f_{i,u} \leq 0.1$.

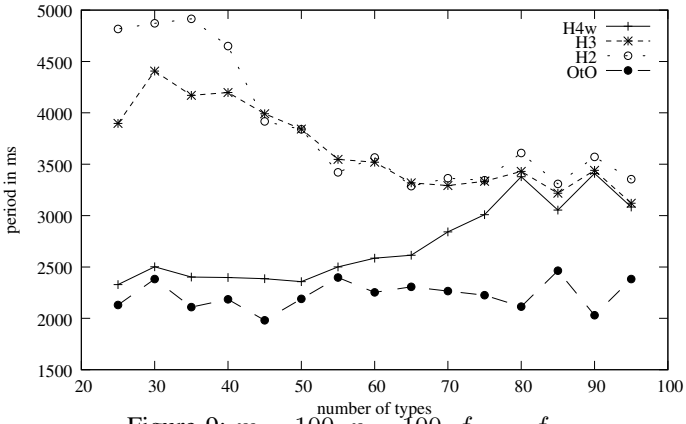

Figure 9: $m = 100$, $n = 100$, $f_{i,u} = f_i$.

C. Comparison with the linear program

This last set of experiments compares our heuristics to the mixed integer linear program (MIP) described in Section VI-A. We restrict the study to small problem instances, so that we are able to derive a solution for the linear program, and results are reported only if 30 successful experiments over 60 trials are obtained with the MIP. Those "MIP-compatible" platforms are selected and the heuristics are run on them.

In the first experiment, we use a platform with 5 machines, and the application has 4 types. We are then able to target applications with up to 15 tasks. Figure 10 shows that H4w is once again the best heuristic but H2 and H4 are close. To measure that difference, Figure 11 shows the normalization of the heuristics with the MIP solution. Results reveal that H2, H3 and H4w are respectively at a factor of 1.73, 1.58 and 1.33 from the optimal.

For the next experiment, we use a platform with 9 machines, and the application has between 5 and 20 tasks of 4 different

types. For visibility reason, we discard results of H1 and H4f from the figure. Figure 12 shows that with more than 15 tasks, the MIP is not able to find solutions anymore.

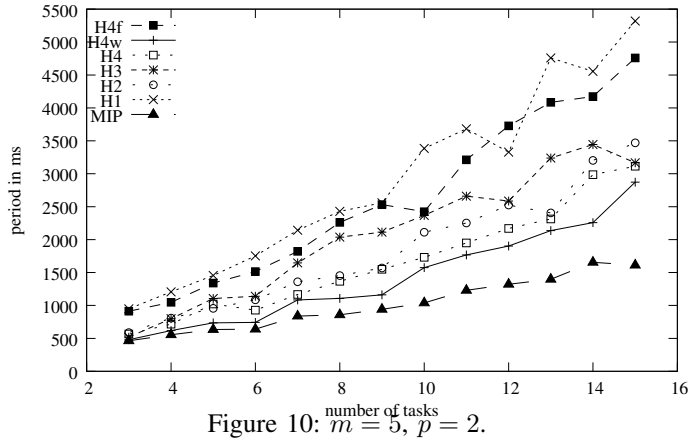

Figure 10: $m = 5$, $p = 2$.

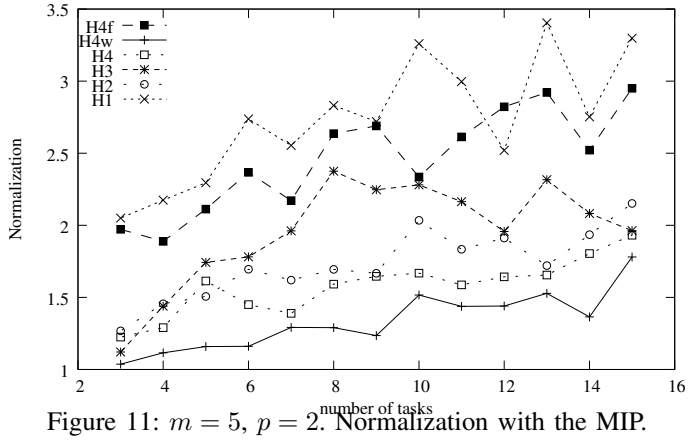

Figure 11: $m = 5$, $p = 2$. Normalization with the MIP.

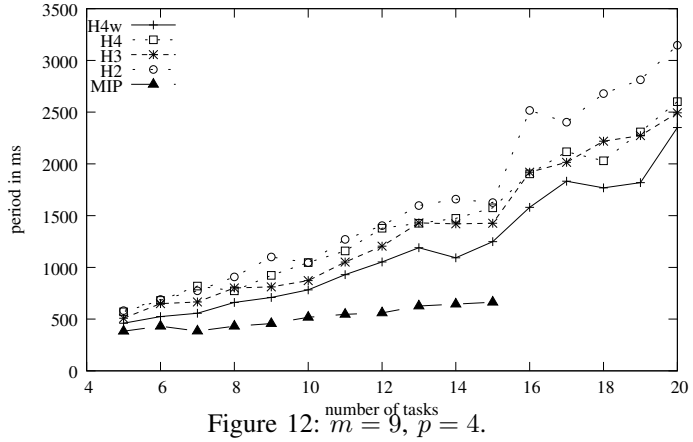

Figure 12: $m = 9$, $p = 4$.

VIII. Conclusion

In this paper, we investigate a throughput optimization problem in the context of micro-factories subject to failures. The problem consists in assigning tasks of tree-shaped application graphs to machines. The failures that occur in the system depend on both the task and the machine on which the task is assigned. We proved that the problem to find an optimal one-to-one mapping for linear chain onto homogeneous machines is polynomial while the problem becomes NP-hard for in-tree one-to-one mappings or for specialized and general mappings. Since general mappings are not usable in practice because of reconfiguration costs, we focused on specialized mappings and proposed several polynomial heuristics to solve the problem when the graph is a linear chain. Also, a mixed linear programming formulation of the problem is given to allow us to evaluate our heuristics by comparing experimental results to the optimal, considering small problem instances. These experimental results showed that the most performing solution is obtained by H4w. This heuristic focuses on the execution speed and does not take into account the failure rate. The comparison between H4w and the optimal solutions that can be found respectively for one-to-one mappings and specialized mappings (onto small platforms thanks to the linear program) showed that H4w is respectively at a factor of 1.28 and 1.33 from the optimal. This is a very promising result, but somehow expected, which means that if we produce fast enough we overcome the faults.

As future work, an interesting problem would be to consider that the instances of a same task can be computed by several machines. Thus, the workload of a task would be divided and the throughput could be improved.

References

[1] A. Benoit, A. Dobrila, J.-M. Nicod, and L. Philippe, "Throughput optimization for micro-factories subject to failures," in *Proc. of the 8th International Symposium on Parallel and Distributed Computing*, Jul. 2009, available as Research Report RR-LIP-2009-02 at graal.ens-lyon.fr/~abenoit/papers/RR-LIP-2009-02.pdf.

[2] M. Tanaka, "Development of desktop machining microfactory," *Journal RIKEN Rev*, vol. 34, pp. 46–49, April 2001, iSSN:0919-3405.

[3] V. P. Nelson, "Fault-tolerant computing: Fundamental concepts," *Computer*, vol. 23, no. 7, pp. 19–25, 1990.

[4] W. Cirne, F. Brasileiro, D. Paranhos, L. F. W. Góes, and W. Voorsluys, "On the efficacy, efficiency and emergent behavior of task replication in large distributed systems," *Parallel Computing*, vol. 33, no. 3, pp. 213–234, 2007.

[5] B. Parhami, "Voting algorithms," *IEEE Transactions on Reliability*, vol. 43, no. 4, pp. 617–629, Dec 1994.

[6] R. West and C. Poellabauer, "Analysis of a window-constrained scheduler for real-time and best-effort packet streams," in *Proc. of the 21st IEEE Real-Time Systems Symp.* IEEE, 2000, pp. 239–248.

[7] R. West, Y. Zhang, K. Schwan, and C. Poellabauer, "Dynamic window-constrained scheduling of real-time streams in media servers," 2004. [Online]. Available: citeseer.ist.psu.edu/article/west03dynamic.html

[8] R. West and K. Schwan, "Dynamic window-constrained scheduling for multimedia applications," in *ICMCS, Vol. 2*, 1999, pp. 87–91. [Online]. Available: citeseer.ist.psu.edu/west98dynamic.html

[9] P. Jalote, *Fault Tolerance in Distributed Systems*. Prentice Hall, 1994.

[10] H. W. Kuhn, "The hungarian method for the assignment problem," *Naval Research Logistics Quarterly*, vol. 2, pp. 83–97, 1955.

[11] I. S. Duff and J. Koster, "On algorithms for permuting large entries to the diagonal of a sparse matrix," *SIAM Journal on Matrix Analysis and Applications*, vol. 22, pp. 973–996, 2001.

[12] M. R. Garey and D. S. Johnson, *Computers and Intractability, a Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, 1979.

[13] "ILOG CPLEX: High-performance software for mathematical programming and optimization," http://www.ilog.com/products/cplex/.